Policy Iteration
v.s.
Value Iteration
v.s.
Prioritized Sweeping

COMP 767 – Reinforcement Learning
January 27th

~Nicolas Angelard-Gontier

• Policy Iteration

Policy iteration (using iterative policy evaluation)

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
   $\quad \Delta \leftarrow 0$
   $\quad$ For each $s \in \mathcal{S}$:
   $\quad\quad v \leftarrow V(s)$
   $\quad\quad V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))\big[r + \gamma V(s')\big]$
   $\quad\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   $\quad$ *old-action* $\leftarrow \pi(s)$
   $\quad \pi(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
   $\quad$ If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

```
def iterative_policy_eval(epsilon=0.1, i=1):
    """
    Policy Evaluation step.
    Iterate through all states to update value function V(s) until the update becomes < epsilon.
    :param epsilon: small positive number to tell when to stop iteration.
    :param i: iteration index
    """

    print "V:", V
    print "Iteration:", i
    print "Number of Bellman updates:", i, "x", len(STATES), "=", i * len(STATES)
    delta = 0
    for s in STATES:
        HISTORY.append(copy.deepcopy(V))  # add deep copy of current Value Function to the history
        v = V[s]  # old state-value
        V[s] = sum([P[s, POLICY[s], s1] * (R[s, POLICY[s], s1] + GAMMA*V[s1]) for s1 in STATES])
        delta = max(delta, abs(v-V[s]))
    if delta >= epsilon:
        return iterative_policy_eval(epsilon, i+1)
    return i
def policy_improvement():
    """
    Policy Improvement step.
    Check if taking any other action yields in a better value.
    If so, change the policy, and return false.
    :return: true only if the policy wasn't changed for all sates.
    """
    policy_stable = True
    for s in STATES:
        current_v = sum([P[s, POLICY[s], s1] * (R[s, POLICY[s], s1] + GAMMA*V[s1]) for s1 in STATES])
        # Taking best action with respect to current value function V:
        for a in ACTIONS:
            temp = sum([P[s, a, s1] * (R[s, a, s1] + GAMMA*V[s1]) for s1 in STATES])
            if temp > current_v:
                POLICY[s] = a  # update policy
                current_v = temp
                policy_stable = False
    return policy_stable
```

Sutton et. al. (RL book)

- # Value Iteration

Value iteration

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat

  $\Delta \leftarrow 0$

  For each $s \in \mathcal{S}$:

    $v \leftarrow V(s)$

    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$

    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi \approx \pi_*$, such that

  $\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$

```
def value_iteration(epsilon=0.1, i=1):
    """
    Just like iterative_policy_eval(), but take the max over all ACTIONS for V(s).
    :param epsilon: small positive number to tell when to stop iteration.
    :param i: iteration index.
    """

    print "V:", V
    print "Iteration:", i
    print "Number of Bellman updates:", i, "x", len(STATES), "=", i * len(STATES)
    delta = 0
    for s in STATES:
        HISTORY.append(copy.deepcopy(V))  # add deep copy of current Value Function to the history
        v = V[s]  # old state-value
        # Taking best action with respect to current value function V:
        for a in ACTIONS:
            temp = sum([P[s, a, s1] * (R[s, a, s1] + GAMMA*V[s1]) for s1 in STATES])
            if temp > V[s]:
                V[s] = temp  # update value function
        delta = max(delta, abs(v-V[s]))
    if delta >= epsilon:
        value_iteration(epsilon, i+1)
```

```
def make_greedy_policy():
    """
    Just like policy_improvement, make policy greedy with respect to V.
    Only difference: this is the ONLY policy update we make since we
    assume that V ~ V*
    """
    policy_improvement()  # make policy greedy with respect to V~V*
```

Sutton et. al. (RL book)

- Prioritize Sweeping

1. Promote state i_{recent} to top of priority queue.
2. While we are allowed further processing and priority queue not empty

   2.1 Remove the top state from the priority queue. Call it i

   2.2 $$\rho_{\text{new}} := \max_{a \in \texttt{actions}(i)} \left(\hat{r}_i^a + \gamma \times \sum_{j \in \texttt{succs}(i,a)} \hat{q}_{ij}^a \hat{J}_j \right)$$

   2.3 $\Delta_{\max} := |\rho_{\text{new}} - \hat{J}_i|$

   2.4 $\hat{J}_i := \rho_{\text{new}}$

   2.5 for each $(i', a') \in \texttt{preds}(i)$

   $P := \hat{q}_{i'i}^{a'} \Delta_{\max}$

   If $P > \epsilon$ (a tiny threshold) and if (i' is not on queue or P exceeds the current priority of i') then promote i' to new priority P.

Andrew W. Moore & Christopher G. Atkeson

```
def prioritized_sweeping():
    """
    Just like value_iteration(), but update states by their priority.
    Start with a priority queue in which we have states that will most change the Value function on the first sweep.
    """
    assert len(P_QUEUE) == len(STATE2PRIORITY) == 0

    # first iteration, we don't have anything in our priority queue, add states that will change in the next sweep:
    for s in STATES:
        newV = copy.deepcopy(V)  # don't update V yet, this is just a simulation to decide what to add in P_QUEUE
        for a in ACTIONS:
            temp = sum([P[s, a, s1] * (R[s, a, s1] + GAMMA * V[s1]) for s1 in STATES])
            if temp > newV[s]:
                newV[s] = temp  # update fake value function
        delta = abs(newV[s] - V[s])
        # add states that will change a lot V in the real sweep:
        if delta > 0:
            heapq.heappush(P_QUEUE, (-delta, s))  # add s with priority -delta (most probable = lower value).
            STATE2PRIORITY[s] = -delta  # keep track of its priority.
    # Iterate over states in the priority queue:
    iteration = 1  # iteration index
    while len(P_QUEUE) > 0:
        HISTORY.append(copy.deepcopy(V))  # add deep copy of current Value Function to the history
        print "V:", V
        print "Iteration:", iteration
        print "Number of Bellman updates:", iteration, "( +", len(STATES), ") =", iteration + len(STATES)
        # print "P_QUEUE:", P_QUEUE
        _, s = heapq.heappop(P_QUEUE)  # pop most probable state.
        del STATE2PRIORITY[s]  # forget its priority.

        # do one Bellman Backup of current state:
        v = V[s]  # old state-value
        for a in ACTIONS:
            temp = sum([P[s, a, s1] * (R[s, a, s1] + GAMMA*V[s1]) for s1 in STATES])
            if temp > V[s]:
                V[s] = temp  # update value function
        delta = abs(v-V[s])

        # add neighbors to the priority queue:
        for s1 in neighbors(s):
            new_priority = - max([delta * P[s1, a, s] for a in ACTIONS])  # how much s1 is influenced by the current change.
            if new_priority < 0:
                # most probable = min value between current and new priority.
                if s1 in STATE2PRIORITY and STATE2PRIORITY[s1] > new_priority:  # update element in priority queue
                    old_priority = STATE2PRIORITY[s1]
                    index = P_QUEUE.index((old_priority, s1))  # current index in the priority queue.
                    P_QUEUE[index] = (new_priority, s1)  # update current priority.
                    STATE2PRIORITY[s1] = new_priority  # keep track of the update.
                elif s1 not in STATE2PRIORITY:  # add new state to priority queue
                    heapq.heappush(P_QUEUE, (new_priority, s1))  # push to priority queue.
                    STATE2PRIORITY[s1] = new_priority  # keep track of its priority.
        iteration += 1
```

Grid World

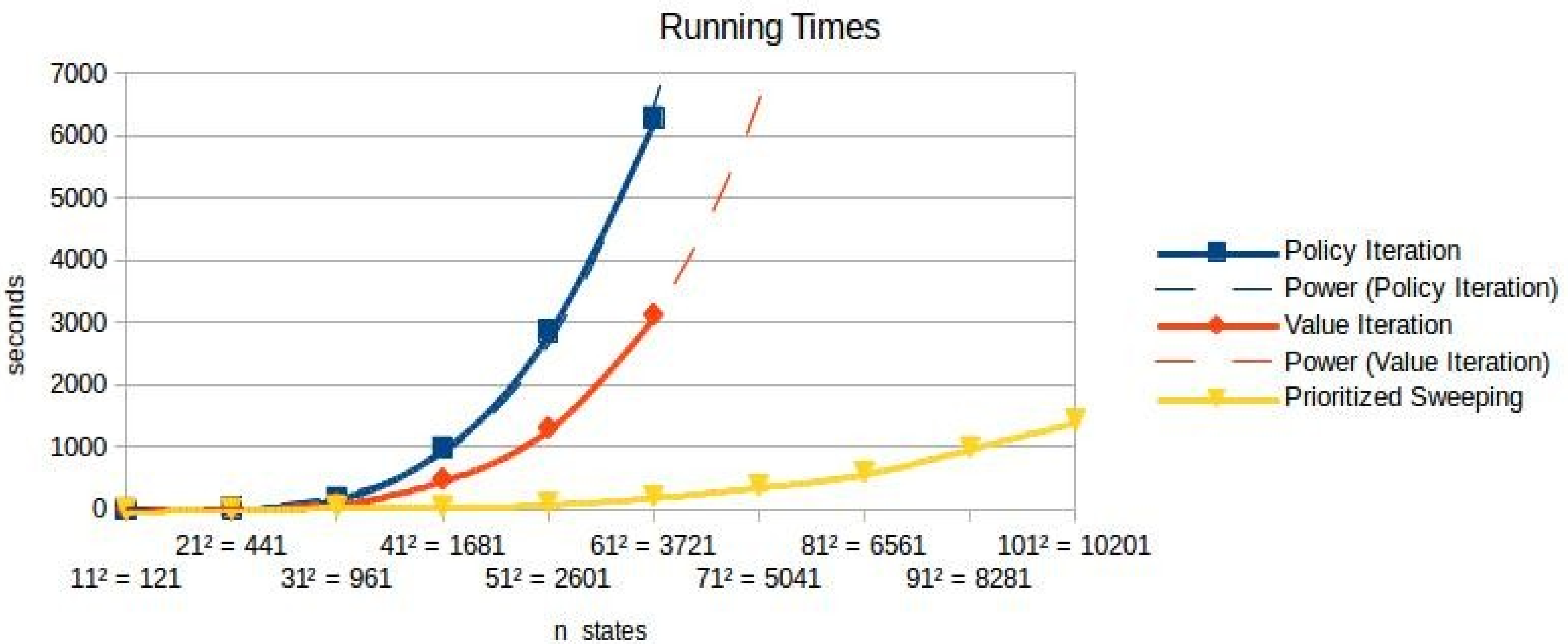
0	0	0	0	0	0	0
0	-10	0	0	0	-10	0
0	0	0	0	0	0	0
0	0	0	100	0	0	0
0	0	0	0	0	0	0
0	-10	0	0	0	-10	0
0	0	0	0	0	0	0

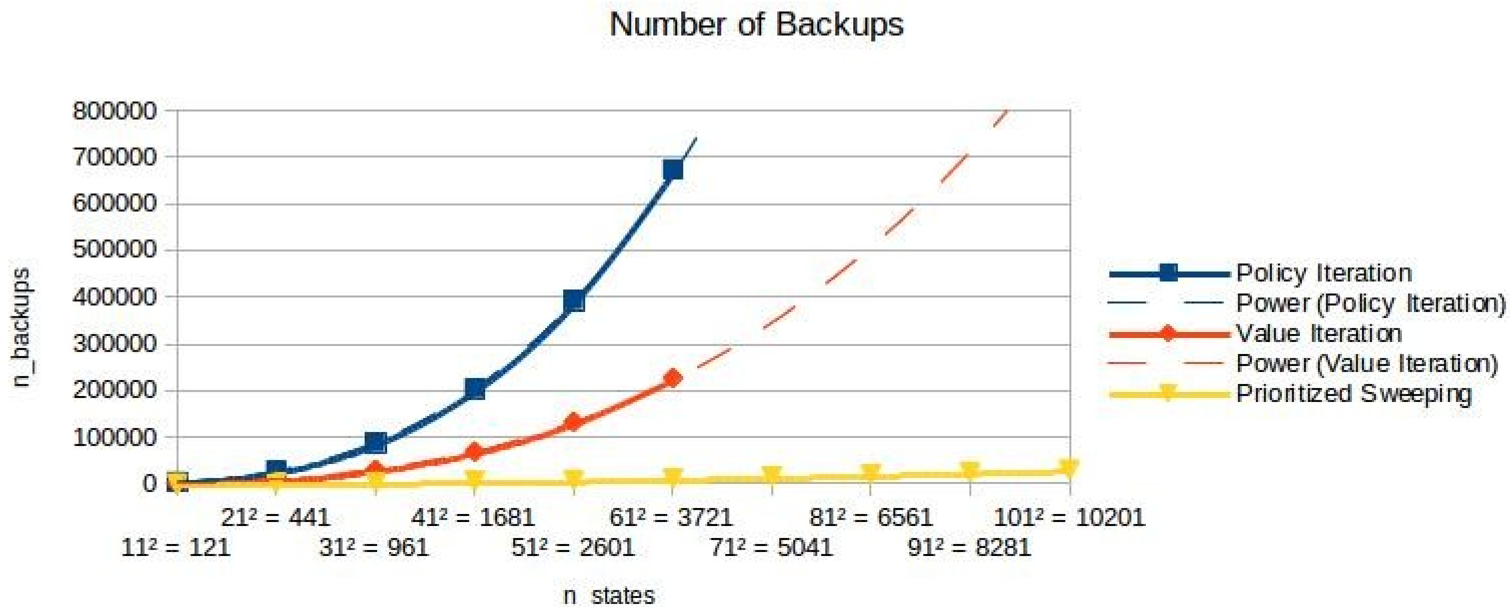
Deterministic grid world example
with width=7 so 49 states:

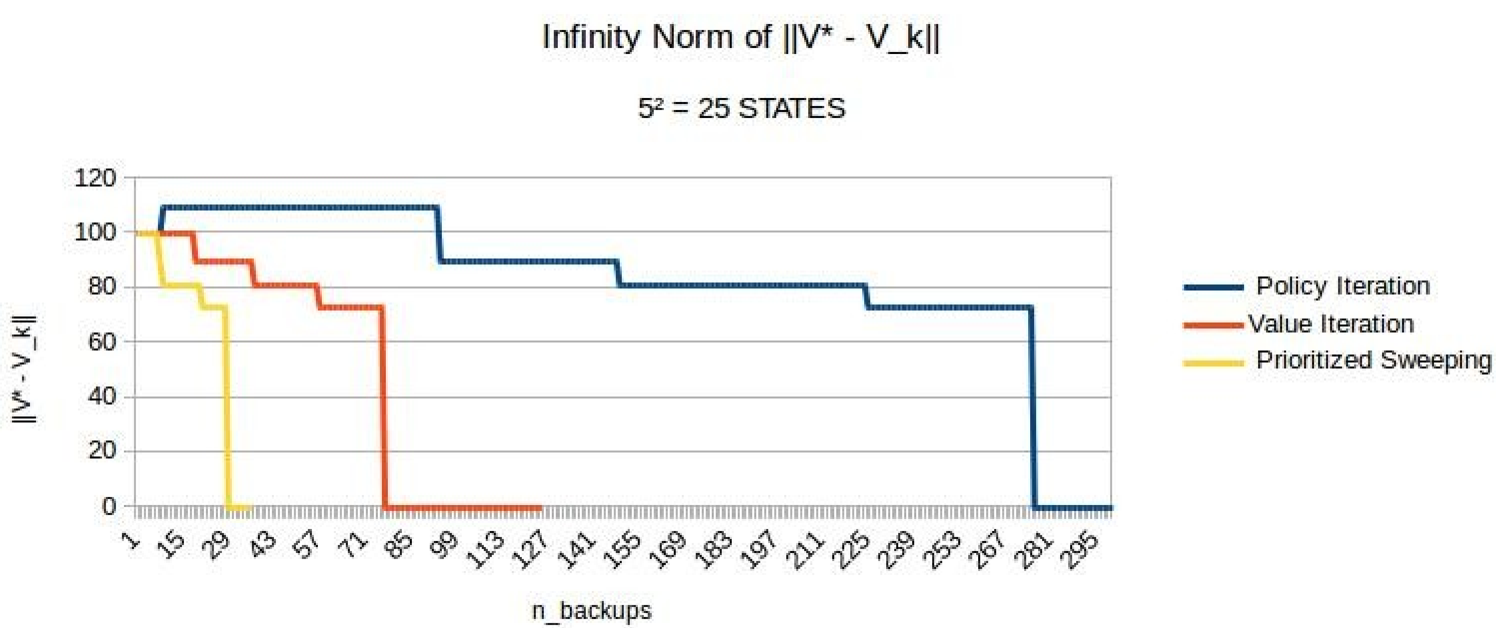
59.049	65.61	72.9	81	72.9	65.61	59.049
65.61	0	81	90	81	0	65.61
72.9	81	90	100	90	81	72.9
81	90	100	0	100	90	81
72.9	81	90	100	90	81	72.9
65.61	0	81	90	81	0	65.61
59.049	65.61	72.9	81	72.9	65.61	59.049

Optimal Value function and Policy
with width=7 so 49 states:

Results


Running Times
7000
6000
5000
4000
3000
2000
1000
0
seconds
11² = 121
21² = 441
31² = 961
41² = 1681
51² = 2601
61² = 3721
71² = 5041
81² = 6561
91² = 8281
101² = 10201
n_states
Policy Iteration
Power (Policy Iteration)
Value Iteration
Power (Value Iteration)
Prioritized Sweeping


Number of Backups
800000
700000
600000
500000
400000
300000
200000
100000
0
n_backups
21² = 441
41² = 1681
61² = 3721
81² = 6561
101² = 10201
11² = 121
31² = 961
51² = 2601
71² = 5041
91² = 8281
n_states
Policy Iteration
Power (Policy Iteration)
Value Iteration
Power (Value Iteration)
Prioritized Sweeping


Infinity Norm of ||V* - V_k||
5² = 25 STATES
120
100
80
60
40
20
0
||V* - V_k||
1
15
29
43
57
71
85
99
113
127
141
155
169
183
197
211
225
239
253
267
281
295
n_backups
Policy Iteration
Value Iteration
Prioritized Sweeping